Breeding Software Test Cases for Complex Systems


National Institute for Systems Test and Productivity
*University of South Florida St Petersburg
**University of South Florida
***University of North Carolina - Asheville
awatkins@stpt.usf.edu

Abstract

The potential cost savings from handling software errors within a development cycle, rather than subsequent cycles, has been estimated at 38.3 billion dollars. Such figures emphasize that current testing methods are inadequate, and that helping reduce software bugs and errors is an important area of research with a substantial payoff. This paper reports on research using genetic algorithms for test case generation for systems level testing, building on past work at the unit testing level. The goals of the paper are to explore the use of genetic algorithms for testing complex distributed systems, as well as to develop a framework or vocabulary of important environmental attributes that characterize complex systems failures. In addition, preliminary visualization techniques that might help software developers to understand and uncover complex systems failures are explored.

1. Introduction

The National Institute of Standards and Technology reported in [1] that users estimated the cost savings to be $11.7 billion if the software they purchased contained 50% fewer bugs and errors. A removal of all bugs in the same development cycle that they were introduced would result in a cost saving of $38.3 billion! These are the cost savings estimated for end users, and there are of course additional savings for the developers. These figures emphasize that current testing methods are inadequate, and that helping reduce software bugs and errors is an important area of research with a substantial payoff.

As software systems become more complicated and increasingly embedded in the processes of business and government, the costs of software failures continue to escalate. Complex distributed system failures are among the most troublesome problems, as the various components interact in unanticipated ways. These highly interconnected systems often suffer from intermittent or transient failures that are particularly difficult to diagnose. Examples of such complex systems include the network-centric warfare architectures being explored by U.S. Department of Defense [2]. These are systems of systems that can be integrated to provide flexible and robust capabilities. However, the failure patterns can be embedded in large, multidimensional spaces of successful executions that are very difficult for humans to test and analyze.

Several approaches to testing have been advocated by researchers and practitioners, including automated test case generation. One of the important benefits of automatically generating test cases is freedom from cognitive bias [3]. When called upon to construct test cases, software developers usually suffer from biases that result in the under exploration of many testing regions. Knowledge of the system internals and operational expectations influence the design of test suites. Automated tools can augment the testing process by providing test cases that are not susceptible to the same biases as the human developers. Combining both human and machine-generated test cases from multiple sources may be the most robust strategy for many testing situations, especially for complex distributed systems.

Genetic algorithms are one technique that has been applied to test case generation with some success. This research builds on past work at the unit testing level [4] and proposes extensions aimed at systems testing. In particular, three main objectives are pursued.

1. While genetic algorithms have shown promise at the unit testing level, one goal of this research is to...
explore such techniques for testing complex distributed systems. The transient, potentially convoluted, failures that often plague distributed systems may best be diagnosed through partially automated approaches that support software developers.

2. In order to characterize complex system failures, a framework or vocabulary of important system-related environmental attributes, such as elapsed time or system load, must be developed. Several such dimensions are used to define the search space explored in this research.

3. Lastly, this research begins to explore visualization techniques that might help software developers uncover and understand complex system failures. While genetic algorithms may produce evidence in the form of test cases, expressed in terms of environmental and system attributes, how that information is presented to developers is a critical issue. Ultimately, a collection of visualization and data mining techniques are likely to be useful in analyzing potential failure patterns.

Several researchers have noted that software testing is very much a model building exercise and that software “testing is applied epistemology” [5]. That is, how we know what we know is an important issue when facing software failures. Software testers must rely on sound research methods to make inferences regarding the possible causes of failures. This paper focuses on using genetic algorithms to produce test cases as a body of evidence for such model building efforts. A prototype complex, distributed system is used to demonstrate the approach. Generated test cases are then analyzed to determine the coverage of the test bed by error and non-error producing test cases. An improvement to the fitness function is made that helps the test data to focus on more even coverage of all errors rather than an in depth coverage of one or two, and a cursory coverage of others. Finally the results of this new fitness function are presented visually, the purpose of this section is to explain how this rich data source can be used to assist the tester in pinpointing errors and to demonstrate the effectiveness of the genetic algorithm.

2. Breeding Test Cases using Genetic Algorithms

Evolutionary computational methods have been of interest to computer scientists, mathematicians, biologists, psychologists, and other investigators since the earliest days of computing research. Mitchell [6] surveys some of the early work conducted on the genetic algorithm. Errors and to demonstrate the effectiveness of the source can be used to assist the tester in pinpointing this new fitness function are presented visually, the space explored in this research. Finally the results of efforts. A prototype complex, distributed system is used to demonstrate the approach. Generated test cases as a body of evidence for such model building exercise. The transient, potentially convoluted, failures that often plague distributed systems may best be diagnosed through partially automated approaches that support software developers.

3. Lastly, this research begins to explore visualization techniques that might help software developers uncover and understand complex system failures. While genetic algorithms may produce evidence in the form of test cases, expressed in terms of environmental and system attributes, how that information is presented to developers is a critical issue. Ultimately, a collection of visualization and data mining techniques are likely to be useful in analyzing potential failure patterns.

Several researchers have noted that software testing is very much a model building exercise and that software “testing is applied epistemology” [5]. That is, how we know what we know is an important issue when facing software failures. Software testers must rely on sound research methods to make inferences regarding the possible causes of failures. This paper focuses on using genetic algorithms to produce test cases as a body of evidence for such model building efforts. A prototype complex, distributed system is used to demonstrate the approach. Generated test cases are then analyzed to determine the coverage of the test bed by error and non-error producing test cases. An improvement to the fitness function is made that helps the test data to focus on more even coverage of all errors rather than an in depth coverage of one or two, and a cursory coverage of others. Finally the results of this new fitness function are presented visually, the purpose of this section is to explain how this rich data source can be used to assist the tester in pinpointing errors and to demonstrate the effectiveness of the genetic algorithm.

2. Breeding Test Cases using Genetic Algorithms

Evolutionary computational methods have been of interest to computer scientists, mathematicians, biologists, psychologists, and other investigators since the earliest days of computing research. Mitchell [6] surveys some of the early work conducted on the genetic algorithm. Errors and to demonstrate the effectiveness of the source can be used to assist the tester in pinpointing this new fitness function are presented visually, the space explored in this research. Finally the results of efforts. A prototype complex, distributed system is used to demonstrate the approach. Generated test cases as a body of evidence for such model building exercise. The transient, potentially convoluted, failures that often plague distributed systems may best be diagnosed through partially automated approaches that support software developers.

3. Lastly, this research begins to explore visualization techniques that might help software developers uncover and understand complex system failures. While genetic algorithms may produce evidence in the form of test cases, expressed in terms of environmental and system attributes, how that information is presented to developers is a critical issue. Ultimately, a collection of visualization and data mining techniques are likely to be useful in analyzing potential failure patterns.

Several researchers have noted that software testing is very much a model building exercise and that software “testing is applied epistemology” [5]. That is, how we know what we know is an important issue when facing software failures. Software testers must rely on sound research methods to make inferences regarding the possible causes of failures. This paper focuses on using genetic algorithms to produce test cases as a body of evidence for such model building efforts. A prototype complex, distributed system is used to demonstrate the approach. Generated test cases are then analyzed to determine the coverage of the test bed by error and non-error producing test cases. An improvement to the fitness function is made that helps the test data to focus on more even coverage of all errors rather than an in depth coverage of one or two, and a cursory coverage of others. Finally the results of this new fitness function are presented visually, the purpose of this section is to explain how this rich data source can be used to assist the tester in pinpointing errors and to demonstrate the effectiveness of the genetic algorithm.

2. Breeding Test Cases using Genetic Algorithms

Evolutionary computational methods have been of interest to computer scientists, mathematicians, biologists, psychologists, and other investigators since the earliest days of computing research. Mitchell [6] surveys some of the early work conducted on the genetic algorithm. Errors and to demonstrate the effectiveness of the source can be used to assist the tester in pinpointing this new fitness function are presented visually, the space explored in this research. Finally the results of efforts. A prototype complex, distributed system is used to demonstrate the approach. Generated test cases as a body of evidence for such model building exercise. The transient, potentially convoluted, failures that often plague distributed systems may best be diagnosed through partially automated approaches that support software developers.

3. Lastly, this research begins to explore visualization techniques that might help software developers uncover and understand complex system failures. While genetic algorithms may produce evidence in the form of test cases, expressed in terms of environmental and system attributes, how that information is presented to developers is a critical issue. Ultimately, a collection of visualization and data mining techniques are likely to be useful in analyzing potential failure patterns.

Several researchers have noted that software testing is very much a model building exercise and that software “testing is applied epistemology” [5]. That is, how we know what we know is an important issue when facing software failures. Software testers must rely on sound research methods to make inferences regarding the possible causes of failures. This paper focuses on using genetic algorithms to produce test cases as a body of evidence for such model building efforts. A prototype complex, distributed system is used to demonstrate the approach. Generated test cases are then analyzed to determine the coverage of the test bed by error and non-error producing test cases. An improvement to the fitness function is made that helps the test data to focus on more even coverage of all errors rather than an in depth coverage of one or two, and a cursory coverage of others. Finally the results of this new fitness function are presented visually, the purpose of this section is to explain how this rich data source can be used to assist the tester in pinpointing errors and to demonstrate the effectiveness of the genetic algorithm.

3. Genetic Algorithms for Unit Testing

Evolutionary algorithms are often used for optimization problems in which the evolution of a population is a search for a satisfactory solution given a set of constraints. In this current application, the genetic algorithm is used to generate good test cases, but the notion of goodness or fitness depends on results from
previous testing cycles. That is, a relative rather than absolute fitness function is used. An example of changing fitness landscapes can be found in the work of [19] with a focus on co-evolution or so called “biological arms races.” For the generation of test cases, a relative fitness function changes along with the population, allowing subsequent generations to take advantage of any insights gleaned from past results stored in the fossil record. This fossil record contains all previously generated test cases and information about the type of error generated, if any, and when that test case was run. Thus, the fossil record provides an environmental context for changing notions of fitness by comparing a test case to previously generated test cases and rewarding the individual based on the following concepts.

1. **Novelty** is a measure of the uniqueness of a particular test case. In the context of method invocations, this can be quantified by measuring the distance in parameter space from previous invocations stored in the fossil record.

2. **Proximity** is a measure of closeness to other test cases that resulted in system failures. This construct requires both the test case description and the execution result that can only be determined by stepping outside the genetic algorithm and executing the test case.

3. **Severity** is measure of the seriousness of a system error. Not all problems are of equal severity. A total system failure or “blue screen of death” is probably worse than a violation of a response time requirement. Deciding on the severity of a spectrum of failure modes allows the system to be customized to particular application domains.

The interplay between these changing factors over successive generations gives rise to surprisingly complex search behaviors that can be characterized in terms of explorers, prospectors, and miners. That is, simple rules, complex behaviors.

The highly iterative search process can best be described in terms of explorers, prospectors, and miners. An explorer is a test case that is highly novel, irrespective of whether it results in success or failure. Essentially, explorers are the test cases that are spread across the lightly populated regions of the test space. Once an error is discovered, the fitness function encourages more thorough testing of the region. Prospectors are test cases that are still somewhat unique, but are also near newly found errors. Therefore, both novelty and proximity combine in the fitness function to generate prospectors. As prospectors uncover additional errors, the fitness function will reward points that are simply near other errors. Miners characterize such test cases that are generated to more fully probe an area in which errors have been previously located. The interplay of these various types of test cases tends to focus attention on error-laden areas, while still generating some points over widespread regions of the test space. Figure 1 is a visualization of the fossil record of test cases for a

![Figure 1. Visualization of a fossil record](image-url)
simple object-oriented program, TRITYP, used by other software researchers. Again, this is a unit testing perspective based on input parameters, for a more complete discussion see [4]. Spikes indicate many test cases in areas where errors were uncovered with other test cases more randomly distributed. In this current project, new attributes are proposed for use in broadening the search space for systems level testing.


Moving from a unit testing to systems testing perspective is one of the goals of this current work. Complex systems fail as the various components interact in unanticipated ways. Examples of such complex systems are the network-centric warfare architectures being explored by U.S. Department of Defense [2]. These are systems of systems that can be integrated to provide flexible and robust capabilities. However, the failure patterns can be embedded in large, multidimensional spaces of successful executions that are very difficult for humans to analyze.

From a systems testing perspective, new environmental attributes become important, for instance [20] identifies the types of failures that can occur when the distributed system fails to respond to a request this is an omission failure, or the response is untimely, a timing failure. Ghosh and Mathur [21] highlight the importance of stress testing systems under varying load levels. Zhang and Cheung [22] focus on stress testing multimedia systems, pointing out that network congestion and transmission errors or delays interrupt data delivery. Building upon these ideas the following list of attributes has been created, along with required input parameters for execution. The system used for research purposes is LOBNET, a test bed developed as a multi-tiered, distributed system that calculates targeting information for simple ballistic weapons.

- **ResponseTime** in milliseconds (ms or msec) or units of one thousandth of a second. This is commonly used to measure disk access times and Internet packet travel times, and therefore, a reasonably natural measure for distributed system execution time. This is used to quantify response time violations and other time-related software failures.

- **ElapsedTime** in hours can be used to identify software failures that occur after prolonged running times. Examples of related faults include memory leaks, counters, fixed-sized arrays, and other bounded structures that may overflow after extended periods. For purposes of this research, the interval [0, 43,800] or roughly 5 years is used.

- **Cyclical temporal units**, such as **DayOfWeek** (7), **HourOfDay** (24), **MonthOfYear** (12) can be used to uncover repetitive failures, such as planned maintenance activities, demanding batch jobs, periodic data loads, or any other scheduled tasks that might coincide with software failures.

- Load factors from the various tiers, such as **AppLoad**, **DBLoad**, **WebLoad**, characterize the computational demands on different components (using a 16 point scale). Excessive stress on any of these tiers may be associated with response time problems or outright failures. Of course, these load factors can be tailored to the target system architecture, but these three provide a basic framework for use in the case study.

- A second set of load factors concentrate on the network layers, including the **WANLoad**, **LANLoad**, and **SANLoad**, again using a 16 point scale. Excessive network traffic can also affect system response time, as well as failure modes. All of these environmental factors can interact during failures. For instance, local network loads may spike upward as a regularly schedule task places demands on system resources.

- System specific input parameters are part of any test case. For the simple LOBNET test bed, there are three parameters, **WeaponID** (5), **DistanceToTarget** in meters [0, 6000], desired **TimeOfImpact** in seconds [0, 86,400] over 24 hours.

- Lastly, the results of an execution include an **ExceptionCode** [0, 8] and a **StackTrace** that introduces some program state into the data set.

These system environment attributes, input parameters, and execution results represent an extended test case search space. The system attributes represent an important resource for identifying patterns in failures. In a sense, the system attributes and calling stack are fragments of program state, making this a gray rather than black box approach. This basic set could obviously be extended in many ways, but provides an initial vocabulary to describe interesting patterns.

The search space is very large, but genetic algorithms have been shown to be a very effective search technique. Holland [7] provided a theoretical basis for the power of genetic algorithms called schema theory, which showed that genetic algorithms derive much of their power from manipulating fit building blocks. The search problem at hand seems to be appropriate, with the system attributes and parameters providing building blocks that combine to partially identify failure patterns. Genetic algorithms may have problems identifying isolated solutions.
surrounded by low fitness regions, basically a “needle in a haystack” [8]. In software failures, this would correspond to a very rare and complicated interaction of many attributes that causes a single failure. While these certainly occur, the bulk of software errors are likely to be less exotic and reasonable targets for genetic algorithm-based approaches. Secondly, finding a needle in a haystack is pretty much a matter of luck for any search technique.

5. LOBNET: A Case Study

A genetic algorithm-based approach was used to generate the test cases. The target system, LOBNET, was designed as a laboratory test bed and mirrors the distributed systems that characterize evolving network-centric warfare components. In order to investigate intermittent failures in such systems, eight errors of varying complexity were injected into LOBNET, using the system environment attributes discussed above, such as elapsed time, load factors, and calendar time. Each of these errors throws an exception code (coded 1-8, with 0 indicating a successful execution). The errors are summarized in Table 1, including the logical conditions for throwing an exception. Along with the exception code, a stack trace is also available and could be used as category label for classification.

Two teams were formed. One team was responsible for seeding the errors, based on combinations of the input parameters and/or system environment attributes. The second team used a genetic algorithm to generate test cases, constructing a database of test results. In this way, the quality of test cases as body of evidence for subsequent analysis and model building can be investigated. The test cases should adequately explore error regions, while still generating enough successful

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values that Throw Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td></td>
</tr>
<tr>
<td>Day</td>
<td>(0,6)</td>
</tr>
<tr>
<td>Hour</td>
<td>(0,23)</td>
</tr>
<tr>
<td>Month</td>
<td>(0,11)</td>
</tr>
<tr>
<td>Elapsed Time</td>
<td>(0,43800)</td>
</tr>
<tr>
<td>Percent of Load</td>
<td></td>
</tr>
<tr>
<td>Application</td>
<td>&gt; 12</td>
</tr>
<tr>
<td>Database</td>
<td>&gt; 13</td>
</tr>
<tr>
<td>Web</td>
<td>&gt; 13</td>
</tr>
<tr>
<td>LAN</td>
<td>13</td>
</tr>
<tr>
<td>SAN</td>
<td>&gt; 10</td>
</tr>
<tr>
<td>SAN</td>
<td>&gt; 10</td>
</tr>
<tr>
<td>WAN</td>
<td>&gt; 10</td>
</tr>
<tr>
<td>Program Parameters</td>
<td></td>
</tr>
<tr>
<td>weapon id</td>
<td>(1,5)</td>
</tr>
<tr>
<td>distance to target</td>
<td>(0,60000)</td>
</tr>
<tr>
<td>time of impact</td>
<td>(0,84600)</td>
</tr>
</tbody>
</table>

Table 1. System Attributes and Parameters with Error Ranges

Figure 2. Amount of rich point test data generated for each error, columns are divided between error test data and neighbor test data.
executions that discriminatory models can be built. One way to assess the search behavior is to visualize the resulting fossil record, contrasting the density of test cases in and around errors with normal executions.

As discussed previously, the fitness of a test case is based on its proximity to errors and other test cases, its novelty and whether it caused an error, the details of the function can be found in [4]. All of the attributes were used in calculating the fitness of a test case. The GA was run for 500 generations, which produced 29,678 unique test cases.

Figure 2 is a breakdown of rich points per error, each is further broken down into whether the test case produced an error or was a neighbor. Rich points are test cases that fall within an error region or are adjacent to that region, while poor points are spread across the remainder of the search space. These results make a strong statement, all the errors were found by the GA, yet there is a disproportionate number of test cases for error six. For reference, error six is the easiest error to find as it covers the largest amount of search space, 19%, while error eight is restricted to 3%. An analysis of the rich to poor ratio for individual errors is shown in Table 2. Due to the overwhelming amount of test data for error six, the other errors have much smaller ratios, four in fact have more poor test data than rich. Although the average ratio is 3.21:1, individually it is not satisfactory, the skew is definitely positive and the kurtosis value shows the distribution to be leptokurtic, a higher peak than normal.

Although successfully finding and exploring all eight errors the results highlight an interesting problem, the GA may focus too intensively on specific error regions. Adequate testing requires that test data not only uncover errors but also have completed a good search of the poor point region. The results displayed demonstrate what is needed are test sets that give good coverage of all errors, not a select few.

### 6. Goldilocks Implementation

A search strategy was devised which uses a breadth first approach to reward novel test cases that cover a larger area of the search space when the ratio is too high, and a depth first approach to build up error test cases when the ratio is too low (emphasizing proximity). Using both strategies to this advantage is not new, [23] proposed a new algorithm using a hybrid of the breadth first and depth first traversal strategies for the construction of binary decision diagrams. Even early expert systems used similar search techniques. For instance, SYNCHEm, a synthetic chemistry expert system used an “effort distribution” algorithm to avoid tunnel vision [24].

To implement the strategy an ongoing count is kept of how many test cases have been unearthed for each error, which is then divided by the total number of error-causing test cases. This percentage is fed into the

<table>
<thead>
<tr>
<th>Table 2. Ratio Mean Rich to Poor</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Error Number</strong></td>
</tr>
<tr>
<td>One</td>
</tr>
<tr>
<td>Two</td>
</tr>
<tr>
<td>Three</td>
</tr>
<tr>
<td>Four</td>
</tr>
<tr>
<td>Five</td>
</tr>
<tr>
<td>Six</td>
</tr>
<tr>
<td>Seven</td>
</tr>
<tr>
<td>Eight</td>
</tr>
</tbody>
</table>

**Mean All Error Ratios**

- Mean: 370.40
- Rich: 170.03
- Poor: 3.21

**Kurtosis**

- Mean: 0.53
- Kurtosis: 2.63

**Skewness**

- Mean: 0.53
- Skewness: 0.65

![Figure 3](image-url)
penalty feature and used to punish newly generated test cases that cause the balance for a particular error to exceed 20%. This percentage rule gives a good balance in allowing an error to thoroughly examine its bounds before pressing the GA to search for more novel test cases. As shown in the previous section, the depth-first search strategy is implemented naturally by the existing fitness function as it attempts to optimize the number of test cases for any error it finds, with the sometimes unfortunate consequence of an unbalanced set of test cases. The breadth-first policy is implemented when errors are temporarily marked as adequately covered, but once the count for a particular error falls below 20% it is encouraged to breed again.

### 8. Test Data Visualization

Generating test data sets that are neighbors of errors is as important as generating test data sets that contain errors, this rich test data helps to define where an error boundary is and assists in analysis to determine what attributes created which errors and at what parameter values. This information will be invaluable in the debugging process. The body of evidence built by the GA can be visualized, statistically analyzed, sliced and diced using online analytic processing (OLAP) tools, and ultimately explored using data mining methods. In this section, several preliminary visualization techniques are investigated.

As the GA evolves it shows a tendency to breed test cases that overlap the rich point region of several errors, as shown in Figure 4 where 39% of the rich points overlapped, compared to 8% generated randomly. The benefit of overlapping is to give a richer test data set from which to mine test data for debugging and retesting. For example, these test data sets can be used to group causes of error, Figure 6 is a web of all the test cases generated for error one, the linkages highlight the neighboring errors (common rich points). There are no links between errors one and two, and errors one and four. A review of the cause of these errors, from Table 1, shows that they have one attribute in common, DAYOFWEEK, and therefore are mutually exclusive.

The histograms give excellent views of the search space as well. Figure 7 shows the three-dimensional view of the distribution. The towers indicate the rich point regions, looking closely you can also see small ridges along day one and day four, these indicate that there is more test data generated for these days than any other. To the software tester unfamiliar with the seeded errors this may indicate that days one and four play a part in more than just the combined errors with HOUROFDAY. One of the causes of the ridge is identified by viewing Figure 8 which is the histogram of DAYOFWEEK and ELAPSEDTIME. The large peak is the neighborhood for error three but the small peak represents no error in this domain, however 55% of the test data generated for this small peak is a neighbor of error one and a further 25% is for error three.

### Table 3. Ratio Mean Rich to Poor

<table>
<thead>
<tr>
<th>Error Number</th>
<th>Mean Rich</th>
<th>Mean Poor</th>
<th>GA Rich: Poor Ratio</th>
<th>Random Generation Rich: Poor Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>One</td>
<td>462.22</td>
<td>160.37</td>
<td>2.88</td>
<td>0.99</td>
</tr>
<tr>
<td>Two</td>
<td>722.33</td>
<td>156.33</td>
<td>4.62</td>
<td>1.03</td>
</tr>
<tr>
<td>Three</td>
<td>41.67</td>
<td>15.90</td>
<td>2.62</td>
<td>0.83</td>
</tr>
<tr>
<td>Four</td>
<td>1100.5</td>
<td>232.89</td>
<td>4.73</td>
<td>0.92</td>
</tr>
<tr>
<td>Five</td>
<td>17.43</td>
<td>6.92</td>
<td>2.52</td>
<td>0.90</td>
</tr>
<tr>
<td>Six</td>
<td>188.22</td>
<td>98.72</td>
<td>1.91</td>
<td>1.01</td>
</tr>
<tr>
<td>Seven</td>
<td>1526.0</td>
<td>379.69</td>
<td>4.02</td>
<td>1.01</td>
</tr>
<tr>
<td>Eight</td>
<td>436.00</td>
<td>243.63</td>
<td>1.79</td>
<td>0.19</td>
</tr>
</tbody>
</table>

Mean All Error Ratios

<table>
<thead>
<tr>
<th>Mean All Error Ratios</th>
<th>561.80</th>
<th>161.81</th>
<th>3.14</th>
<th>0.86</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kurtosis</td>
<td>-0.05</td>
<td>-0.09</td>
<td>1.65</td>
<td>6.75</td>
</tr>
<tr>
<td>Skewness</td>
<td>0.89</td>
<td>0.41</td>
<td>0.40</td>
<td>-2.55</td>
</tr>
</tbody>
</table>

Figure 3 shows the distribution of the rich points with the implemented Goldilocks strategy. The scale on the chart is the same as in Figure 2 to illustrate the positive change in the distribution. The rich to poor ratios for all error is displayed in Table 3. The counts for each error still vary, however the variation is much more acceptable, no single error gets the lion’s share of the search space.

### 7. Comparison to Random Generation

Test case generation by the GA can be compared to another common automated technique, random generation. As [25] points out random testing is inexpensive as all that is required is a random number generator and it is more stressing to the system than manual generation. Research by [26] also concluded that random testing was a useful validation tool. The downside to random generation is the lack of full coverage of a system. Our goal in this research has been to create test cases that generate errors and additional test cases in the neighborhood of errors. How do the results of the random generator compare? The test generator was run another 500 generations with all the features of the GA removed, comparison charts are shown in Figures 4 and 5, 71% of the test sets generated by the GA were error cases, compared to 14% generated randomly, the rich to poor ratios are shown in the last column of Table 3.
Genetic algorithms are a powerful optimization tool, but they have proved themselves equally adept in an environment where they are used to generate test data. The LOBNET experiments presented here have demonstrated that the GAs can generate test data sets at a systems testing level, with a focus on error regions of the code. An improvement has been made to the fitness function to make the number of test cases found for each error more uniform. The test data sets should be viewed as a rich resource allowing one to reproduce.

**Figure 4.** (left) The breakdown of the generated test data, 14% are poor points. Of the rich points, 37% of the test data sets are errors and not a neighbor to another error, while 34% are both errors and neighbors.

**Figure 5.** (right) The breakdown of the randomly generated test data, 62% are poor points. Of the rich points, 10% of the test data sets are errors and not a neighbor to another error, while 4% are both errors and neighbors.

**Figure 6.** This web shows the overlap of rich points from error one, for example errors one and six have 1159 rich points in common. Errors one and two and one and four are mutually exclusive and share no errors.

**9. Conclusion**

Genetic algorithms are a powerful optimization tool, but they have proved themselves equally adept in an environment where they are used to generate test data. The LOBNET experiments presented here have demonstrated that the GAs can generate test data sets at a systems testing level, with a focus on error regions of the code. An improvement has been made to the fitness function to make the number of test cases found for each error more uniform. The test data sets should be viewed as a rich resource allowing one to reproduce.
errors when testing and retesting, but also can be used to determine where in the code errors might exist by using histograms and distributions graphs. Additionally data mining tools can be used to determine the rules that caused the errors. This latter use is an area for future research.

A limitation to the research presented here is that the seeded errors were all expressed in terms of the system attributes in our framework. Currently, we are focusing on introducing noise, via exceptions that are only partially expressed using these attributes. This will more realistically simulate a real-world environment. Of course, there is no substitute for field use. A software-testing tool being developed under the auspices of the National Institute for Systems Test and Productivity at the University of South Florida will provide an interesting opportunity to apply these techniques in the field.

![Histogram of Test Data Distribution](image1)

**Figure 7.** The histogram of test data distribution for errors one and two. The large peaks are the errors and their neighbors.

![Histogram of Test Cases Day of Week vs Elapsed Time](image2)

**Figure 8.** The histogram of test data distribution for Week and Elapsed Time. The large peak is the collection of rich points for Error 4.
References